Shanghai Jiaotong University
Machine Learning

# Project Report

## Classify CIFAR-10 and MNIST Datasats through Different Techniques Using Matlab

Name: Hong Yining ID: 515021910453 Date: 13 November 2017

## Abstract

The goal of this project is to devise different machine learning classifiers and compare the advantages and disadvantages of them.The datasets to be classified are CIFAR-10 and MNIST. In this paper, five different classifiers are introduced and implemented. The features, derivation and methods of classifiers are discussed. In addition to the classifiers introduced in classes, kernel models are also carried out.The classifiers are: KNN, linear SVM, kernel SVM, Fisher's Linear Discriminant and Kernel Fisher Discriminant. The whole project is written in Matlab and the running results are given. Classifiers are compared with regard to accuracy, easiness to implement, training and testing time, memory requirement and so on.

**Keywords: Machine Learning, CIFAR-10, MNIST, classifier, KNN, SVM, Fisher, Matlab**

# 1 Introduction

## 1.1 Motivation

The goal of this project is to devise machine learning classifiers that perform well on CIFAR-10 and MNIST datasets. CIFAR-10 dataset contains 50000 labeled $32 \times 32$ RGB images in 10 classes and are divided into five data batches. MNIST dataset contains 60000 labeled $28 \times 28$ images in 10 classes. To classify the images, I utilized different machine learning techniques: k-Nearest-Neighbors classifier(KNN); Support Vector Machine classifier (SVM), including linear SVM and kernel SVM; Fisher classifier, including Fisher's Linear Discriminant(also called FLD) and Kernel Fisher Discriminant (KFD). For CIFAR-10 datasets, features were extracted with the VLFeat toolboxs HOG feature function, while for MNIST, I did not extract any features. Using my techinques, I managed to achieve high accuracies using all five approaches on both datasets.

## 1.2 Background and Related Work

The CIFAR-10 dataset is a subset of another dataset, called the 80 million tiny images data set. Each CIFAR-10 image has been labeled with a class, from a choice of: airplane, automobile, bird, cat, deer, dog, frog, horse, ship, truck. This dataset has several advantages, especially for a novice to machine learning. Since the images are only 32x32 pixels large, the default feature set is relatively small, and so it is far easier to process these images with simple algorithms, limited time, and meager computational power. Additionally, since there are only 10 classes, and they tend to be quite diverse, it is somewhat easier to obtain high accuracies than on a database like CIFAR-100, which has 100 class labels.

The MNIST database of handwritten digits, has a training set of 60000 examples, and a test set of 10000 examples. It is a subset of a larger set available from NIST. The digits have been size-normalized and centered in a fixed-size image.It is a good database for people who want to try learning techniques and pattern recognition methods on real-world data while spending minimal efforts on preprocessing and formatting.

Before beginning work on our algorithms, I researched state-of-the-art approaches to solving the CIFAR-10 and MNIST classification problem. Almost universally, the best solutions used some variation of convolutional neural net, such as Lee and Xie's 2015 paper Deeply-Supervised Nets, which reported an accuracy of 91.78 % on CIFAR-10, and Ciresan reported an accuracy of 99.77% using convolutional networks on MNIST. However, CNN has not been taught yet in our classes. Previous work reveals that KNN and SVM outshine other classifiers in many cases. For example, Belongie had an accuracy of 99.37% using KNN on MNIST, while DeCoste and Scholkopf achieved an accuracy of 99.46% using SVM on MNIST. Moreover, Prof.Ni has repeated several times in our classes that the best classifiers this century include KNN and SVM. Thus, I decided to implement KNN and SVM in this project. Meanwhile, Prof.Ni uploaded an additional material about Fisher Linear Discriminant in our group as an additional material for LDA, which aroused my interest. I decided to classify with Fisher LDA.

While I was browsing through papers concerning different machine learning techniques, I noted that the word "kernel" has frequently appeared. I searched online and found that kernel is a method which enable people to operate in a high-dimensional, implicit feature space without ever computing the coordinates of the data in that space, but rather by simply computing the inner products between the images of all pairs of data in the feature space. It seemed as a challenge to me. I tried to implement it using Matlab, and with the help of multiple resources, I managed to devise kernel algorithms.

However, as we all know, kernel is more like a "trick", which means that we need to apply it carefully. I actually read an interview of Yann LeCun, who is not very interested in kernel trick. He said that "Basically, varying the smoothness of the kernel function allows us to interpolate between two simple methods: linear classification, and template matching. I got in trouble about 10 years ago by saying that kernel methods were a form of glorified template matching." He said CNN can be a more useful way to compute highly complex function with a limited amount of computation. However, since we haven't learnt CNN yet, I decided that kernel is worth trying.

# 2 Method Review

## 2.1 Preprocessing

### 2.1.1 Load MNIST images and labels

The files downloaded from Prof.Yann LeCun's website are binary files. For some reasons, the files cannot be identified by Matlab. I read the lecture notes of Prof.Andrew Ng from Stanford University, and used ***loadMNISTImages.m*** and ***loadMNISTLabels.m*** to read my MNIST datasets.

The key idea of the two scripts is to first get fid using fopen, and then to move downward to get the number of images/labels, sizes and so on, then read the images/labels after that. After getting all the data, we use reshape function of Matlab to reshape the data to dimensions we want.

Note that the original images loaded are stored as col $\times$ row $\times$ imagenumber(since images are loaded one by one). However, I want my matrix to be 60000 $\times$ 784. Thus, I used permute and reshape function of Matlab to turn it into 60000 $\times$ 784 matrix.

### 2.1.2 Feature Extraction

For CIFAR10, I intended to use some feature extraction features. However, for MNIST, feature extraction is unnecessary.

Since the images are only 32$\times$32 pixels, I initially took each pixel as a feature for training the classifiers for a total of 3072 features. I soon found that it was useful to run feature extraction before running the classifiers on the image data, and to train the classifiers on the features of the images rather than the raw image data. I converted images into new feature sets using HOG feature extraction.

HOG stands for histogram of oriented gradients, and is a very commmon technique in image feature extraction. HOG splits the image up into regions, and calculates the frequencies of appearances of various gradient directions within each region. The final feature set is simply the list of all these gradient frequencies.

I used the VLFeat library to convert the image into vectorized HOG descriptors. Note that VLFeat toolbox should be installed in advance in my computer. **The *init.m* should be run first every time you open Matlab before training and testing CIFAR-10.** HOG features are extracted through *naivehog.m*. Therefore, in *train.m* and *classify.m*, we change the images to features by using naivehog function.

## 2.2 Classifiers

**Notice:**

- If you want to run on CIFAR-10, you should first run *init.m*. Also, ***train.m*** **and** ***classify.m*****are for choosing the classifiers you want(etc. KNN, SVM...)**.

- If you want to run on MNIST, just choose the classifiers yo want in ***localhost2.m*** and run it. **If the memory of your computer is less than 8BG, run *localhost3.m* instead if you want to try SVM linear, SVM kernel or Fisher kernel, or else there might be an "out of memory" error.** (in the case of "out of memory", I divide the dataset into 6 small subsets, each containing 10000 images).

- In case the relative path cannot be used and data batches cannot be uploaded, in *init.m* and all *localtest.m* you can set the path manually by adding something into path=";

### 2.2.1 K-nearest-neighbours

The KNN classifier rule is to classify **x** by assigning it the label most frequently represented among the k nearest samples and use a voting scheme. What is shown below is the case when k equals to 5.and we can see that the black points are more than white points in the range. So **x** is classified to the black kind.

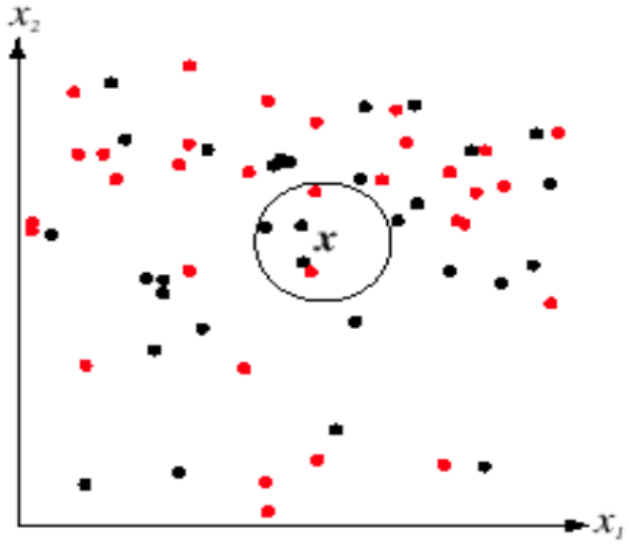


Figure 1: KNN classifier

KNN works by comparing an image to every image in the training set, then taking the k-closest images from the training, and classifying as the most common label in the k-closest. In my implementation of KNN, I calculated the distance between an image in the test batch to all images in the training batch, selected the k-smallest distances, and then selected the label from the training sample who's k-closest to the test sample.

### 2.2.2 linear SVM

Support vector machine is a supervised learning model with associated learning algorithms that analyze data used for classification and regression analysis. Given a set of training examples, each marked as belonging to one or the other of two categories, an SVM training algorithm builds a model that assigns new examples to one category or the other, making it a non-probabilistic binary linear classifier. An SVM model is a representation of the examples as points in space, mapped so that the examples of the separate categories are divided by a clear gap that is as wide as possible. New examples are then mapped into that same space and predicted to belong to a category based on which side of the gap they fall.

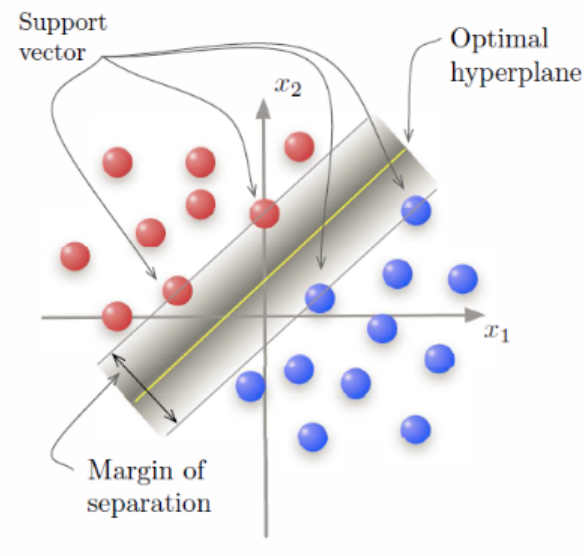


Figure 2: SVM linear classifier

Given $N$ training data points:

$$\mathbf{x_1}, ..., \mathbf{x_i}, .., \mathbf{x_N} \in \Re^{\mathbf{m}}$$

each with a label of:

$$d_i = +1 \text{ or } d_i = -1$$

find a hyperplane:

$$\mathbf{w^T x + b = 0} \text{ ( } \mathbf{w} \in \Re^{\mathbf{m}}, \text{ b} \in \Re \text{ )}$$

that saparate data into two groups:

$$d_i = +1$$
$$d_i = -1$$

we want to find $w_0$ and $b_0$ that define the optimal hyperplane

$$g(\mathbf{x}) = \mathbf{w_0^T x + b_0 = 0}$$

so that for training $\mathbf{x_i}$

$$g((\mathbf{x_i})) = \mathbf{w_0^T x_i + b_0} \geq \mathbf{+1} \text{ for } d_i = +1$$
$$g((\mathbf{x_i})) = \mathbf{w_0^T x_i + b_0} \leq \mathbf{+1} \text{ for } d_i = -1$$

We can put this together to get the optimization problem:

"Minimize $\|\mathbf{w}\|\|\mathbf{w}\|$ subject to $y_i(\mathbf{w} \cdot \mathbf{x}_i - b) \geq 1$,$y_i(\mathbf{w} \cdot \mathbf{x}_i - b) \geq 1$,$for i = 1, \ldots, n i = 1, \ldots, n$"

### 2.2.3 Kernel SVM

The kernel trick avoids the explicit mapping that is needed to get linear learning algorithms to learn a nonlinear function or decision boundary. For all $\mathbf{x}$ and $\mathbf{x}'$ in the input space $\mathcal{X}$, certain functions $k(\mathbf{x}, \mathbf{x}')$ can be expressed as an inner product in another space $\mathcal{V}$. The function $k\colon \mathcal{X} \times \mathcal{X} \to \mathcal{R}$ is often referred to as a kernel or a kernel function.

Certain problems in machine learning have additional structure than an arbitrary weighting function $k$. The computation is made much simpler if the kernel can be written in the form of a "feature map" $\varphi\colon \mathcal{X} \to \mathcal{V}$ which satisfies $k(\mathbf{x}, \mathbf{x}') = \langle \varphi(\mathbf{x}), \varphi(\mathbf{x}') \rangle_{\mathcal{V}}$. The key restriction is that $\langle \cdot, \cdot \rangle_{\mathcal{V}}$ must be a proper inner product. On the other hand, an explicit representation for $\varphi$ is not necessary, as long as $\mathcal{V}$ is an inner product space. The alternative follows from Mercer's theorem: an implicitly defined function $\varphi$ exists whenever the space $\mathcal{X}$ can be equipped with a suitable measure ensuring the function$k$ k satisfies Mercer's condition.

Mercer's condition can be reduced to this simpler case. If we choose as our measure the counting measure $\mu(T) = |T|$ for all $T \subset X$, which counts the number of points inside the set $T$, then the integral in Mercer's theorem reduces to a summation:

$$\sum_{i=1}^{n} \sum_{j=1}^{n} k(\mathbf{x}_i, \mathbf{x}_j) c_i c_j \geq 0.$$

If this summation holds for all finite sequences of points $(\mathbf{x}_1, \ldots, \mathbf{x}_n)$ in $\mathcal{X}$ and all choices of n real-valued coefficients $(c_1, \ldots, c_n)$, then the function $k$ satisfies Mercer's condition.

Some algorithms that depend on arbitrary relationships in the native space $\mathcal{X}$ would, in fact, have a linear interpretation in a different setting: the range space of $\varphi$ . The linear interpretation gives us insight about the algorithm. Furthermore, there is often no need to compute $\varphi$ directly during computation, as is the case with support vector machines. Some cite this running time shortcut as the primary benefit. Researchers also use it to justify the meanings and properties of existing algorithms.

Theoretically, a Gram matrix $\mathbf{K} \in \mathcal{R}^{n \times n}$ with respect to $\{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$, where $K_{ij} = k(\mathbf{x}_i, \mathbf{x}_j)$, must be positive semi-definite (PSD). Empirically, for machine learning heuristics, choices of a function $k$ that do not satisfy Mercer's condition may still perform reasonably if $k$ at least approximates the intuitive idea of similarity. Regardless of whether $k$ is a Mercer kernel, $k$ may still be referred to as a "kernel".

If the kernel function $k$ is also a covariance function as used in Gaussian processes, then the Gram matrix $\mathbf{K}$ can also be called a covariance matrix. Finally, suppose that $\mathbf{K}$ is a square matrix. Then $\mathbf{K}^{\mathrm{T}}\mathbf{K}$ is a positive-semi-definite matrix.

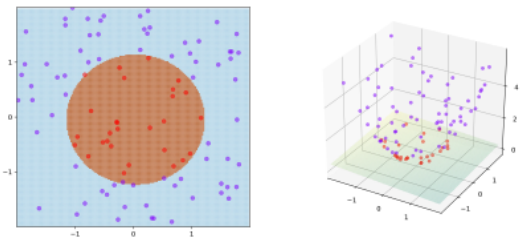

Figure 3: SVM kernel classifier

SVM with kernel given by $\phi((a, b)) = (a, b, a^2, b^2)$ and thus $K(x, y) = xy + x^2y^2$. The training points are mapped to a 3-dimensional space where a separating hyperplane can be easily found.

The SVM separation task is now acting on in the transformed space to find the support vectors that generate

$$h\Phi(x) + b = \pm 1$$

for the hypothesis vector $h = \sum c_i \Phi(x_i)$ given by the sum over support vector points $x_i$. Putting both expressions together we get

$$\sum c_i K(x_i, x) + b = \pm 1$$

with the scalar kernel function $K(x_i, x) = \Phi(x_i)\Phi(x)$. The kernel is composed out of the scalar product between a support vector $x_i$ and another feature vector point $x$ in the transformed space.

### 2.2.4 Fisher's Linear Discriminant

Intuitively, the idea of LDA is to find a projection where class separation is maximized. Given two sets of labeled data, $\mathbf{C}_1$ and $\mathbf{C}_2$, define the class means $\mathbf{m}_1$ and $\mathbf{m}_2$ to be

$$\mathbf{m}_i = \frac{1}{l_i} \sum_{n=1}^{l_i} \mathbf{x}_n^i,$$

where $l_i$ is the number of examples of class $\mathbf{C}_i$ . The goal of linear discriminant analysis is to give a large separation of the class means while also keeping the in-class variance small.[4] This is formulated as maximizing, with respect to $\mathbf{w}$ , the following ratio:

$$J(\mathbf{w}) = \frac{\mathbf{w}^{\mathrm{T}} \mathbf{S}_B \mathbf{w}}{\mathbf{w}^{\mathrm{T}} \mathbf{S}_W \mathbf{w}},$$

where $\mathbf{S}_B$ is the between-class covariance matrix and $\mathbf{S}_W$ is the total within-class covariance matrix:

$$\mathbf{S}_B = (\mathbf{m}_2 - \mathbf{m}_1)(\mathbf{m}_2 - \mathbf{m}_1)^{\mathrm{T}}$$
$$\mathbf{S}_W = \sum_{i=1,2} \sum_{n=1}^{l_i} (\mathbf{x}_n^i - \mathbf{m}_i)(\mathbf{x}_n^i - \mathbf{m}_i)^{\mathrm{T}}.$$

The maximum of the above ratio is attained at

$$\mathbf{w} \propto \mathbf{S}_W^{-1}(\mathbf{m}_2 - \mathbf{m}_1).$$

as can be shown by the Lagrange multiplier method (sketch of proof):

Maximizing $J(\mathbf{w}) = \dfrac{\mathbf{w}^{\mathrm{T}} \mathbf{S}_B \mathbf{w}}{\mathbf{w}^{\mathrm{T}} \mathbf{S}_W \mathbf{w}}$ is equivalent to maximizing

$\mathbf{w}^{\mathrm{T}} \mathbf{S}_B \mathbf{w}$

subject to

$\mathbf{w}^{\mathrm{T}} \mathbf{S}_W \mathbf{w} = 1.$

This, in turn, is equivalent to maximizing $I(\mathbf{w}, \lambda) = \mathbf{w}^{\mathrm{T}} \mathbf{S}_B \mathbf{w} - \lambda(\mathbf{w}^{\mathrm{T}} \mathbf{S}_W \mathbf{w} - 1)$ , where $\lambda$ is the Lagrange

multiplier.

At the maximum, the derivatives of $I(\mathbf{w}, \lambda)$ with respect to $\mathbf{w}$ and $\lambda$ must be zero. Taking $\frac{dI}{d\mathbf{w}} = \mathbf{0}$ yields.

$$\mathbf{S}_B\mathbf{w} - \lambda\mathbf{S}_W\mathbf{w} = \mathbf{0},$$

which is trivially satisfied by $\mathbf{w} = c\mathbf{S}_W^{-1}(\mathbf{m}_2 - \mathbf{m}_1)$ and $\lambda = (\mathbf{m}_2 - \mathbf{m}_1)^{\mathrm{T}}\mathbf{S}_W^{-1}(\mathbf{m}_2 - \mathbf{m}_1)$.

### 2.2.5 Kernel Fisher Discriminant

To extend LDA to non-linear mappings, the data, given as the $\ell$ points $\mathbf{x}_i$, can be mapped to a new feature space, $F$, via some function $\phi$. . In this new feature space, the function that needs to be maximized is

$$J(\mathbf{w}) = \frac{\mathbf{w}^{\mathrm{T}}\mathbf{S}_B^{\phi}\mathbf{w}}{\mathbf{w}^{\mathrm{T}}\mathbf{S}_W^{\phi}\mathbf{w}},$$

where

$$\mathbf{S}_B^{\phi} = \left(\mathbf{m}_2^{\phi} - \mathbf{m}_1^{\phi}\right)\left(\mathbf{m}_2^{\phi} - \mathbf{m}_1^{\phi}\right)^{\mathrm{T}}$$
$$\mathbf{S}_W^{\phi} = \sum_{i=1,2}^{} \sum_{n=1}^{l_i} \left(\phi(\mathbf{x}_n^i) - \mathbf{m}_i^{\phi}\right)\left(\phi(\mathbf{x}_n^i) - \mathbf{m}_i^{\phi}\right)^{\mathrm{T}},$$

and

$$\mathbf{m}_i^{\phi} = \frac{1}{l_i}\sum_{j=1}^{l_i}\phi(\mathbf{x}_j^i).$$

Further, note that $\mathbf{w} \in F$. Explicitly computing the mappings $\phi(\mathbf{x}_i)$ and then performing LDA can be computationally expensive, and in many cases intractable. For example, $F$ may be infinitely dimensional. Thus, rather than explicitly mapping the data to $F$, the data can be implicitly embedded by rewriting the algorithm in terms of dot products and using the kernel trick in which the dot product in the new feature space is replaced by a kernel function, $k(\mathbf{x}, \mathbf{y}) = \phi(\mathbf{x}) \cdot \phi(\mathbf{y})$ . LDA can be reformulated in terms of dot products by first noting that $\mathbf{w}$ will have an expansion of the form $\mathbf{w} = \sum_{i=1}^{l} \alpha_i\phi(\mathbf{x}_i)$. Then note that

$$\mathbf{w}^{\mathrm{T}}\mathbf{m}_i^{\phi} = \frac{1}{l_i}\sum_{j=1}^{l}\sum_{k=1}^{l_i}\alpha_j k\left(\mathbf{x}_j, \mathbf{x}_k^i\right) = \alpha^{\mathrm{T}}\mathbf{M}_i,$$

where

$$(\mathbf{M}_i)_j = \frac{1}{l_i}\sum_{k=1}^{l_i} k(\mathbf{x}_j, \mathbf{x}_k^i).$$

The numerator of $J(\mathbf{w})$ can then be written as:

$$\mathbf{w}^{\mathrm{T}}\mathbf{S}_B^{\phi}\mathbf{w} = \mathbf{w}^{\mathrm{T}}\left(\mathbf{m}_2^{\phi} - \mathbf{m}_1^{\phi}\right)\left(\mathbf{m}_2^{\phi} - \mathbf{m}_1^{\phi}\right)^{\mathrm{T}}\mathbf{w} = \alpha^{\mathrm{T}}\mathbf{M}\alpha, \qquad \text{where} \qquad \mathbf{M} = (\mathbf{M}_2 - \mathbf{M}_1)(\mathbf{M}_2 - \mathbf{M}_1)^{\mathrm{T}}.$$

Similarly, the denominator can be written as

$$\mathbf{w}^{\mathrm{T}}\mathbf{S}_W^{\phi}\mathbf{w} = \alpha^{\mathrm{T}}\mathbf{N}\alpha, \qquad \text{where} \qquad \mathbf{N} = \sum_{j=1,2}\mathbf{K}_j(\mathbf{I} - \mathbf{1}_{l_j})\mathbf{K}_j^{\mathrm{T}},$$

with the $n^{\text{th}}, m^{\text{th}}$ component of $\mathbf{K}_j$ defined as $k(\mathbf{x}_n, \mathbf{x}_m^j)$, $\mathbf{I}$ is the identity matrix, and $\mathbf{1}_{l_j}$ the matrix with all entries equal to $1/l_j$ . This identity can be derived by starting out with the expression for $\mathbf{w}^{\mathrm{T}}\mathbf{S}_W^{\phi}\mathbf{w}$ and using the expansion of $\mathbf{w}$ and the definitions of $\mathbf{S}_W^{\phi}$ and $\mathbf{m}_i^{\phi}$

$$
\begin{aligned}
\mathbf{w}^{\mathrm{T}}\mathbf{S}_W^{\phi}\mathbf{w} &= \left(\sum_{i=1}^{l}\alpha_i\phi^{\mathrm{T}}(\mathbf{x}_i)\right)\left(\sum_{j=1,2}\sum_{n=1}^{l_j}\left(\phi(\mathbf{x}_n^j)-\mathbf{m}_j^{\phi}\right)\left(\phi(\mathbf{x}_n^j)-\mathbf{m}_j^{\phi}\right)^{\mathrm{T}}\right)\left(\sum_{k=1}^{l}\alpha_k\phi(\mathbf{x}_k)\right) \\
&= \sum_{j=1,2}\sum_{i=1}^{l}\sum_{n=1}^{l_j}\sum_{k=1}^{l}\left(\alpha_i\phi^{\mathrm{T}}(\mathbf{x}_i)\left(\phi(\mathbf{x}_n^j)-\mathbf{m}_j^{\phi}\right)\left(\phi(\mathbf{x}_n^j)-\mathbf{m}_j^{\phi}\right)^{\mathrm{T}}\alpha_k\phi(\mathbf{x}_k)\right) \\
&= \sum_{j=1,2}\sum_{i=1}^{l}\sum_{n=1}^{l_j}\sum_{k=1}^{l}\left(\alpha_i k(\mathbf{x}_i,\mathbf{x}_n^j)-\frac{1}{l_j}\sum_{p=1}^{l_j}\alpha_i k(\mathbf{x}_i,\mathbf{x}_p^j)\right)\left(\alpha_k k(\mathbf{x}_k,\mathbf{x}_n^j)-\frac{1}{l_j}\sum_{q=1}^{l_j}\alpha_k k(\mathbf{x}_k,\mathbf{x}_q^j)\right) \\
&= \sum_{j=1,2}\left(\sum_{i=1}^{l}\sum_{n=1}^{l_j}\sum_{k=1}^{l}\left(\alpha_i\alpha_k k(\mathbf{x}_i,\mathbf{x}_n^j)k(\mathbf{x}_k,\mathbf{x}_n^j)-\frac{2\alpha_i\alpha_k}{l_j}\sum_{p=1}^{l_j}k(\mathbf{x}_i,\mathbf{x}_n^j)k(\mathbf{x}_k,\mathbf{x}_p^j)+\frac{\alpha_i\alpha_k}{l_j^2}\sum_{p=1}^{l_j}\sum_{q=1}^{l_j}k(\mathbf{x}_i,\mathbf{x}_p^j)k(\mathbf{x}_k,\mathbf{x}_q^j)\right)\right) \\
&= \sum_{j=1,2}\left(\sum_{i=1}^{l}\sum_{n=1}^{l_j}\sum_{k=1}^{l}\left(\alpha_i\alpha_k k(\mathbf{x}_i,\mathbf{x}_n^j)k(\mathbf{x}_k,\mathbf{x}_n^j)-\frac{\alpha_i\alpha_k}{l_j}\sum_{p=1}^{l_j}k(\mathbf{x}_i,\mathbf{x}_n^j)k(\mathbf{x}_k,\mathbf{x}_p^j)\right)\right) \\
&= \sum_{j=1,2}\alpha^{\mathrm{T}}\mathbf{K}_j\mathbf{K}_j^{\mathrm{T}}\alpha-\alpha^{\mathrm{T}}\mathbf{K}_j\mathbf{1}_{l_j}\mathbf{K}_j^{\mathrm{T}}\alpha \\
&= \alpha^{\mathrm{T}}\mathbf{N}\alpha.
\end{aligned}
$$

With these equations for the numerator and denominator of$J(\mathbf{w})$ , the equation for $J$ can be rewritten as

$$J(\alpha)=\frac{\alpha^{\mathrm{T}}\mathbf{M}\alpha}{\alpha^{\mathrm{T}}\mathbf{N}\alpha}.$$

Then, differentiating and setting equal to zero gives

$$(\alpha^{\mathrm{T}}\mathbf{M}\alpha)\mathbf{N}\alpha=(\alpha^{\mathrm{T}}\mathbf{N}\alpha)\mathbf{M}\alpha.$$

Since only the direction of $\mathbf{w}$ , and hence the direction of $\alpha$, matters, the above can be solved for $\alpha$ as

$$\alpha=\mathbf{N}^{-1}(\mathbf{M}_2-\mathbf{M}_1).$$

Note that in practice, $\mathbf{N}$ is usually singular and so a multiple of the identity is added to it

$$\mathbf{N}_{\epsilon}=\mathbf{N}+\epsilon\mathbf{I}.$$

Given the solution for $\alpha$ , the projection of a new data point is given by

$$y(\mathbf{x})=(\mathbf{w}\cdot\phi(\mathbf{x}))=\sum_{i=1}^{l}\alpha_i k(\mathbf{x}_i,\mathbf{x}).$$

# 3 Comparison and Discussion

The final result is like this:

| Stage | KNN | SVM linear | SVM kernel | FLD | KFD |
|---|---|---|---|---|---|
| CIFAR-10 | 0.561880 | 0.530400 | 0.638240 | 0.551840 | 0.641740 |
| MNIST | 0.9665 | 0.908333 | 0.965617 | 0.8652 | 0.947200 |

Table 1: Accuracies of different classifiers on CIFAR-10 and MNIST

From Table 1, we can see that KNN works fine on both datasets, especially MNIST. In MNIST, it scores 96.65%, which is the highest among the five classifiers. Moreover, KNN is trivial and easy to implement. However, in KNN, all of the work is done in the testing phase, training the KNN is as simple as just saving the training data, training labels, and model parameters, and setting k. Because of this, the KNN model is very large and takes a long time to test. And due to the long testing time, I just set k to 30 and never change it later. I think by changing k, KNN can score better.

In my project, SVM linear does not have a sound performance. I think it might be due to overfitting of my model. Also, SVM takes a long time to train and has high requirement for the memory of computer. My computer has 8 GB, however, I was reminded that it cannot compute a $60000 \times 60000$ matrix. I think it's because the algorithm relied heavily on the Octave Quadratic Programming solver (qp) which was prohibitively slow and has high time complexity. Therefore, I divide MNIST into 6 small sets. This might reduce the accuracy(the reduction is lower than 2%). I also divide MNIST into small sets in SVM kernel and KFD.

While I was testing on Fisher's Linear Discriminant, the result was given much more quickly than the other classifiers. I think it's because the dimensions are reduced. Thus, the computation is largely simplified, making the classifier faster than all the others. However, FLD scored badly on both datasets. I think it's because FLD relies heavily on the means of the samples, and the datasets have more information about variance rather than means. Also, overfitting might cause the low accuracy.

One surprising result is that if we use kernel, the accuracy can be boosted a lot, and the time consumed is not long compared to other clasiffiers. For example, SVM kernel and KFD both scores above 63% on CIFAR-10, much higher than other classifiers. I think it's because features of more dimensions are applied by kernel, but the complex computation in higher dimensions are saved due to kernel function. Particularly, in Fisher kernel, every value has to compute with variance and means, plus weight. Thus, fisher vector does not only contain features in one dimension, but contains various structural information while building the model.

I also found that HOG feature extraction helped a lot for me. For example, without HOG, KNN can only score an accuracy of around 30%. By extracting HOG feature, the accuracy is improved by around 30%.

# 4 Conclusion

In this project, I implemented five different classifiers, including linear and kernel classifiers. The results are all acceptable. In CIFAR-10, KFD scored a best accuracy of 64.17%; while in MNIST, KNN scored a best accuracy of 96.65%. I implemented my classifiers quite early. However, debugging and testing and improving take lots of time.

I encountered multiple problems in this project. One is that when I finished writing my algorithms, I found

it worked fine on CIFAR-10, however, all classifiers scored less than 10% in MNIST. I turned to Prof.Ni for help, and he told me that there's no need to do feature extraction in MNIST. Nevertheless, classifiers still cannot work with MNIST after I remove feature extraction. I thought since classifiers function well with CIFAR-10, there must be problems with loading the images rather than the classifiers themselves. And I finally found the problem - when I load MNIST images, the images were read one by one and thus need to be reshaped to $28 \times 28 \times 60000$ matrix. However, I initially thought MNIST must have the same format as CIFAR-10 and reshape the images to $60000 \times 28 \times 28$ matrix. The data was disrupted by this.

The other problem is that I was always reminded "out of memory" in Matlab. This is because $60000 \times 60000$ is indeed a large matrix and requires a memory of $\geq 11$ GB. My computer is 64-bit and has a memory of 8GB. I tried to install some memory banks in my computer but I failed. I finally managed with dividing the datasets into 6 small datasets. And the five CIFAR-10 data batches are trained and tested independently, and the average of accuracies are calculated. This might reduce the accuracy by within 2.5%, but the accuracies is acceptable.

I still have a lot to do in the future. For example, due to time limits, I didn't test KNN with different k and initially set k as 30. I will try different k and get more desirable results. Moreover, I'll retry with my memory banks and see what's going on. With more memory, I'll be able to train and test the data batches all at once and the accuracy can be boosted.

And, most importantly, there are still many classifiers for me to carry out, and learn, and compare them with the ones I carried out.

# 5 reference


Learning Multiple Layers of Features from Tiny Images, Alex Krizhevsky, 2009.

Deeply-supervised nets, Chen-yu LEE, 2015

High performance convolutional neural networks for image classification, Dan C. Ciresan, 2010

http://www.vlfeat.org/overview/hog.html

http://cs231n.github.io/classification/#summaryapply

http://blog.sciencenet.cn/blog-56590-884346.html

https://github.com/jbuckman/10601-f15-proj

http://yann.lecun.com/exdb/mnist/

https://en.wikipedia.org/wiki/Kernel_method https://www.youtube.com/watch?v=P2Ew4Ljyi6Y

http://jmlr.org/proceedings/papers/v38/lee15a.pdf

http://blog.csdn.net/tracer9/article/details/51253604

http://blog.csdn.net/happyer88/article/details/46576379

http://www.cs.toronto.edu/~rsalakhu/papers/srivastava14a.pdf

# 6 Appendix

## 6.1 Appendix I: Codes

### 6.1.1 main programs for running classifiers

Codes of localtest.m in Matlab

```
%this is the script for test your classifier locally, replace the path with your setup for the given dataset and enjoy~
clear;
path = '';
digits 4;
A = [];
for i = 1 : 5
A = [A,load([path,'data_batch_',num2str(i),'.mat'])];
end
B = load([path,'test_batch.mat']);
accuracy = 0;
for i = 1 : 5
traindata = [];
trainlabels = [];
disp(['Training Batch ',num2str(i)]);
traindata = A(i).data;
trainlabels = A(i).labels;
testdata = B.data;
testlabels = B.labels;
Model = train(traindata,trainlabels);
disp(['Testing Batch ',num2str(i)]);
resultlabels = classify(Model,testdata);
disp(['Accuracy is ', num2str(sum(resultlabels == testlabels) / 10000)]);
accuracy = accuracy + sum(resultlabels == testlabels);
end
accuracy = accuracy / 50000;
fprintf('Total accuracy = %f\n',accuracy);
```

Codes of localtest2.m in Matlab

```
clear;
path = '';
digits 4;

A = loadMNISTImages([path,'train-images.idx3-ubyte']);
B = loadMNISTLabels([path,'train-labels.idx1-ubyte']);
C = loadMNISTImages([path,'t10k-images.idx3-ubyte']);
D = loadMNISTLabels([path,'t10k-labels.idx1-ubyte']);
accuracy = 0;
testdata = C;
testlabels = D;

traindata = A;
trainlabels = B;
disp(['Training Batch ']);
Model = trainKNN(traindata,trainlabels);
%Model = trainFisher(traindata,trainlabels);
```

```
        %Model = trainSVM_linear(traindata,trainlabels);
        %Model = trainSVM_QuadKernel(traindata,trainlabels);
        %Model = trainFisherwKernel(traindata,trainlabels,@poly2,2000);
        disp(['Testing Batch ']);
        resultlabels = KNNclassify(Model,testdata);
        %resultlabels = FisherClassify(Model,testdata);
        %resultlabels = SVM_linearClassify(Model,testdata);
        %resultlabels = SVM_QuadKernelClassify(Model,testdata);
        %resultlabels = FisherwKernelClassify(Model,testdata,@poly2);
        disp(['Accuracy is ', num2str(sum(resultlabels == testlabels) / 10000)
            ]);
```

Codes of localtest3.m in Matlab

```
        clear;
        path = '';
        digits 4;

        A = loadMNISTImages([path,'train-images.idx3-ubyte']);
        B = loadMNISTLabels([path,'train-labels.idx1-ubyte']);
        C = loadMNISTImages([path,'t10k-images.idx3-ubyte']);
        D = loadMNISTLabels([path,'t10k-labels.idx1-ubyte']);
        accuracy = 0;
        testdata = C;
        testlabels = D;

        for i = 1 : 6
        traindata = [];
        trainlabels = [];
        disp(['Training Batch ',num2str(i)]);
        siz = size(A,1)/6;
        siz1 = siz * (i-1) + 1;
        siz2 = siz * i;
        for j = siz1 : siz2
        traindata = [traindata; A(j,:)];
        trainlabels = [trainlabels; B(j,:)];
        end
        %Model = trainSVM_linear(traindata,trainlabels);
        %Model = trainSVM_QuadKernel(traindata,trainlabels);
        Model = trainFisherwKernel(traindata,trainlabels,@poly2,2000);
        disp(['Testing Batch ',num2str(i)]);
        %resultlabels = SVM_linearClassify(Model,testdata);
        %resultlabels = SVM_QuadKernelClassify(Model,testdata);
        resultlabels = FisherwKernelClassify(Model,testdata,@poly2);
        disp(['Accuracy is ', num2str(sum(resultlabels == testlabels) / 10000)
            ]);
        accuracy = accuracy + sum(resultlabels == testlabels);
        end
        accuracy = accuracy / 60000;
        fprintf('Total accuracy = %f\n',accuracy);
```

#### 6.1.2 train

Codes of train.m in Matlab

```
function [Model] = train(X,Y)
data = [];
for i = 1 : size(X,1)
data = [data;naivehog(reshape(X(i,:),[32,32,3]),8)'];
%disp(i);
end
%Model = trainSVM_QuadKernel(data,Y);
%Model = trainKNN(data,Y);
%Model = trainSVM_linear(data,Y);
Model = trainFisherwKernel(data,Y,@poly2,2000);
%Model = trainFisher(data,Y);
end
```

Codes of trainKNN.m in Matlab

```
function [ Model ] = trainKNN(datas,labels)
Model.datas = datas;
Model.labels = labels;
Model.k = 30;
end
```

Codes of trainFisher.m in Matlab

```
function Model = trainFisher(data,label,epsilon)
if nargin < 3
epsilon = 0.001;
end

data = double(data);
m = mean(data);
n = max(label)-min(label) + 1;
cdata = bsxfun(@plus,data,-mean(data));
St = cdata'*data;
Sb = zeros(size(data,2),size(data,2));
Model.Centers = [];
for i = min(label) : max(label)
cdata = data(label == i,:);
mc = mean(cdata);
Sb = Sb + size(cdata,2)*(mc-m)'*(mc-m);
Model.Centers = [Model.Centers;mc];
end
Sw = St-Sb;
Sw = Sw + epsilon*eye(size(Sw));
[V,~] = eig(Sw\Sb);
Model.W = V(:,1:n-1);
Model.Centers = Model.Centers * Model.W;
end
```

Codes of trainFisherwKernelr.m in Matlab

```
function Model = trainFisherwKernel(data,label,kernel,epsilon)
if nargin < 4
epsilon = 0.001;
end

data = double(data);
```

```
        l = size(data,1);
        c = max(label)-min(label)+1;
        kmatrix = kernel(data,data);
        Ms = sum(kmatrix,2) ./ l;
        M = zeros(size(Ms,1),size(Ms,1));
        N = zeros(size(M));
        for i = min(label) : max(label)
        ki = kmatrix(:,label == i);
        li = size(ki,2);
        Mi = sum(ki,2);
        M = M + (Mi-Ms) * (Mi-Ms)';
        N = N + ki * (eye(li)-ones(li)./li) * ki';
        end
        N = N + epsilon*eye(size(N));
        [V,D] = eig(N\M);
        [~,idx] = sort(diag(D),'descend');
        Model.center = [];
        Model.alpha = V(:,idx(1:c-1));
        Model.traindata = data;

        for i = min(label) : max(label)
        ki = kmatrix(label == i,:);
        cc = ki * Model.alpha;
        cc = mean(cc);
        Model.center = [Model.center;cc];
        end

        end
```

Codes of trainSVM_linear.m in Matlab

```
        function Model=trainSVM_linear(data,label,C)
        label=single(label);
        if nargin < 3
        C=0.1;
        end
        numClass=max(label)-min(label)+1;
        Model.W=[];
        Model.b=[];
        for i=1:numClass
        label1=label;
        label1(label==i-1,1)=1;
        label1(label~=i-1,1)=-1;
        [w,b]=binarySVM_linear(data,label1,C);
        Model.W=[Model.W;w];
        Model.b=[Model.b;b];
        end
        end
        function [w,b]=binarySVM_linear(data,label,C)
        H=(data*data').*(label*label');
        f=ones(size(label,1),1);
        Aeq=label';
        beq=0;
        lb=zeros(size(label,1),1);
        ub=zeros(size(label,1),1)+C;
```

```
25          alpha=quadprog(double(H),-1*f,[],[],double(Aeq),beq,lb,ub,[],optimset(
                'Algorithm','interior-point-convex','Display','off'));
26          w=(alpha.*label)'*data;
27          sv_idx=find((alpha>1e-5)&(alpha<C-1e-5));
28          b_all=1./label-(w*data')';
29          b=mean(b_all(sv_idx));
30          end
```

Codes of trainSVM_QuadKernel.m in Matlab

```
1           function Model=trainSVM_QuadKernel(data,label,C)
2           label=single(label);
3           if nargin < 3
4           C=0.1;
5           end
6           numClass=max(label)-min(label)+1;
7           Model.SV=cell(numClass,1);
8           Model.alpha_sv=cell(numClass,1);
9           Model.label_sv=cell(numClass,1);
10          Model.b=[];
11          for i=1:numClass
12          label1=label;
13          label1(label==i-1,1)=1;
14          label1(label~=i-1,1)=-1;
15          [SV,alpha_sv,label_sv,b]=binarySVM_QuadKernel(data,label1,C);
16          Model.SV{i}=SV;
17          Model.alpha_sv{i}=alpha_sv;
18          Model.label_sv{i}=label_sv;
19          Model.b=[Model.b;b];
20          end
21          end
22          function [SV,alpha_sv,label_sv,b]=binarySVM_QuadKernel(data,label,C)
23          kernel=(data*data'+1).^2;
24          H=kernel.*(label*label');
25          f=ones(size(label,1),1);
26          Aeq=label';
27          beq=0;
28          lb=zeros(size(label,1),1);
29          ub=zeros(size(label,1),1)+C;
30          alpha=quadprog(double(H),-1*f,[],[],double(Aeq),beq,lb,ub,[],optimset(
                'Algorithm','interior-point-convex','Display','off'));
31          sv_idx=find(alpha>1e-5);%&(alpha<C-1e-4));
32          sv_idx1=find((alpha>1e-4)&(alpha<C-1e-4));
33          SV=data(sv_idx,:);
34          alpha_sv=alpha(sv_idx,:);
35          label_sv=label(sv_idx,:);
36          b_sv=(1./label(sv_idx1,:))'-(alpha.*label)'*kernel(:,sv_idx1);
37          b=mean(b_sv);
38          end
```

### 6.1.3 classify

Codes of classify.m in Matlab

```
1           function Y = classify(Model,X)
```

```
        data = [];
        for i = 1 : size(X,1)
        data = [data;naivehog(reshape(X(i,:),[32,32,3]),8)'];
        end
        %Y = SVM_linearClassify(Model,data);
        %Y = KNNclassify(Model,data);
        %Y = SVM_QuadKernelClassify(Model,data);
        Y = FisherwKernelClassify(Model,data,@poly2);
        %Y = FisherClassify(Model,data);
        end
```

Codes of FisherClassify.m in Matlab

```
        function labels = FisherClassify(Model,datas)
        datas = datas * Model.W;
        dist = pdist2(datas,Model.Centers);
        [~,labels] = min(dist,[],2);
        labels = labels - ones(size(labels));
        end
```

Codes of FisherwKernelClassify.m in Matlab

```
        function [ labels ] = FisherwKernelClassify(Model,datas,kernel)
        M = kernel(datas,Model.traindata);
        datas = M * Model.alpha;
        dist = pdist2(datas,Model.center);
        [~,labels] = min(dist,[],2);
        labels = labels - ones(size(labels));
        end
```

Codes of KNNclassify.m in Matlab

```
        function [labels] = KNNclassify(Model,datas)
        labels = [];
        for i = 1 : size(datas,1)
        dist = pdist2(datas(i,:),Model.datas);
        [~,idx] = sort(dist,'ascend');

        labels = [labels;mode(Model.labels(idx(1:Model.k)))];
        end
        end
```

Codes of SVM_linearClassify.m in Matlab

```
        function labels=SVM_linearClassify(Model,datas)
        margins = datas * (Model.W)'+ repmat((Model.b)',[size(datas,1),1]);
        [~,labels] = max(margins,[],2);
        labels = labels - ones(size(labels));
        end
```

Codes of SV_QuadKernelClassify.m in Matlab

```
        function labels=SVM_QuadKernelClassify(Model,datas)
        numClass=size(Model.b,1);
        margins=zeros(size(datas,1),numClass);
        for i=1:numClass
        kernel=(Model.SV{i}*datas'+1).^2;
```

```
        margins(:,i)=((Model.alpha_sv{i}.*Model.label_sv{i})'*kernel)'+ Model.
            b(i,1);
        end
        [~,labels] = max(margins,[],2);
        labels = labels - ones(size(labels));
        end
```

#### 6.1.4 preprocessing

Codes of SV_init.m in Matlab

```
        %This code is for the initialization for local vlfeat
        %Please replace the path with your vlfeat setup path

        %path = '';
        %run([path,'\vlfeat-0.9.20\toolbox\vl_setup.m']);
        run([which('vl_setup.m')]);
```

Codes of naivehog.m in Matlab

```
        function [ hog ] = naivehog(I,cellSize)
        if nargin<2
        cellSize = 8;
        end
        I = im2single(I);
        hog = vl_hog(I,cellSize);
        hog = hog(:);
        end
```

Codes of loadMNISTImages.m in Matlab

```
        function images = loadMNISTImages(filename)
        %loadMNISTImages returns a 28x28x[number of MNIST images] matrix
            containing
        %the raw MNIST images

        fp = fopen(filename, 'rb');
        assert(fp ~= -1, ['Could not open ', filename, '']);

        magic = fread(fp, 1, 'int32', 0, 'ieee-be');
        assert(magic == 2051, ['Bad magic number in ', filename, '']);

        numImages = fread(fp, 1, 'int32', 0, 'ieee-be');
        numRows = fread(fp, 1, 'int32', 0, 'ieee-be');
        numCols = fread(fp, 1, 'int32', 0, 'ieee-be');

        images = fread(fp, inf, 'unsigned char');
        images = reshape(images, numCols, numRows, numImages);
        images = permute(images,[3 2 1]);

        fclose(fp);

        % Reshape to #pixels x #examples
        images = reshape(images, size(images, 1), size(images, 2) * size(
            images, 3));
```

```
23          % Convert to double and rescale to [0,1]
24          images = double(images) / 255;
25
26          end
```

Codes of loadMNISTLabels.m in Matlab

```
1           function labels = loadMNISTLabels(filename)
2           %loadMNISTLabels returns a [number of MNIST images]x1 matrix
                containing
3           %the labels for the MNIST images
4
5           fp = fopen(filename, 'rb');
6           assert(fp ~= -1, ['Could not open ', filename, '']);
7
8           magic = fread(fp, 1, 'int32', 0, 'ieee-be');
9           assert(magic == 2049, ['Bad magic number in ', filename, '']);
10
11          numLabels = fread(fp, 1, 'int32', 0, 'ieee-be');
12
13          labels = fread(fp, inf, 'unsigned char');
14
15          assert(size(labels,1) == numLabels, 'Mismatch in label count');
16
17          fclose(fp);
18
19          end
```

## 6.2 Appendix II: Running Results

```
>> localtest
Training Batch 1
Testing Batch 1
Accuracy is 0.5624
Training Batch 2
Testing Batch 2
Accuracy is 0.5655
Training Batch 3
Testing Batch 3
Accuracy is 0.5606
Training Batch 4
Testing Batch 4
Accuracy is 0.5629
Training Batch 5
Testing Batch 5
Accuracy is 0.558
Total accuracy = 0.561880
```

Figure 4: KNN on CIFAR-10

```
Training Batch
Testing Batch
Accuracy is 0.9665
```

Figure 5: KNN on MNIST

```
>> localtest
Training Batch 1
Testing Batch 1
Accuracy is 0.5341
Training Batch 2
Testing Batch 2
Accuracy is 0.5302
Training Batch 3
Testing Batch 3
Accuracy is 0.5284
Training Batch 4
Testing Batch 4
Accuracy is 0.5316
Training Batch 5
Testing Batch 5
Accuracy is 0.5277
Total accuracy = 0.530400
>>
```

Figure 6: SVM LINEAR on CIFAR-10

```
Testing Batch 1
Accuracy is 0.9077
Training Batch 2
Testing Batch 2
Accuracy is 0.9082
Training Batch 3
Testing Batch 3
Accuracy is 0.9088
Training Batch 4
Testing Batch 4
Accuracy is 0.909
Training Batch 5
Testing Batch 5
Accuracy is 0.9087
Training Batch 6
Testing Batch 6
Accuracy is 0.9076
Total accuracy = 0.908333
```

Figure 7: SVM LINEAR on MNIST

```
>> localtest
Training Batch 1
Testing Batch 1
Accuracy is 0.6351
Training Batch 2
Testing Batch 2
Accuracy is 0.6375
Training Batch 3
Testing Batch 3
Accuracy is 0.6419
Training Batch 4
Testing Batch 4
Accuracy is 0.6409
Training Batch 5
Testing Batch 5
Accuracy is 0.6358
Total accuracy = 0.638240
```

Figure 8: SVM kernel on CIFAR-10

```
Training Batch 1
Testing Batch 1
Accuracy is 0.9648
Training Batch 2
Testing Batch 2
Accuracy is 0.9681
Training Batch 3
Testing Batch 3
Accuracy is 0.9681
Training Batch 4
Testing Batch 4
Accuracy is 0.9664
Training Batch 5
Testing Batch 5
Accuracy is 0.9631
Training Batch 6
Testing Batch 6
Accuracy is 0.9632
Total accuracy = 0.965617
```

Figure 9: SVM kernel on MNIST

```
>> localtest
Training Batch 1
Testing Batch 1
Accuracy is 0.5478
Training Batch 2
Testing Batch 2
Accuracy is 0.5529
Training Batch 3
Testing Batch 3
Accuracy is 0.5539
Training Batch 4
Testing Batch 4
Accuracy is 0.553
Training Batch 5
Testing Batch 5
Accuracy is 0.5516
Total accuracy = 0.551840
```

Figure 10: FLD on CIFAR-10

```
>> localtest2
Training Batch
Testing Batch
Accuracy is 0.8652
```

Figure 11: FLD on MNIST

```
>> localtest
Training Batch 1
Testing Batch 1
Accuracy is 0.6422
Training Batch 2
Testing Batch 2
Accuracy is 0.6378
Training Batch 3
Testing Batch 3
Accuracy is 0.6465
Training Batch 4
Testing Batch 4
Accuracy is 0.6421
Training Batch 5
Testing Batch 5
Accuracy is 0.6401
Total accuracy = 0.641740
```

Figure 12: KFD on CIFAR-10

```
Training Batch 1
Testing Batch 1
Accuracy is 0.9466
Training Batch 2
Testing Batch 2
Accuracy is 0.9465
Training Batch 3
Testing Batch 3
Accuracy is 0.9496
Training Batch 4
Testing Batch 4
Accuracy is 0.9506
Training Batch 5
Testing Batch 5
Accuracy is 0.9428
Training Batch 6
Testing Batch 6
Accuracy is 0.9471
Total accuracy = 0.947200
```

Figure 13: KFD on MNIST